**What are SHAP values?**

SHAP stands for Shapley Additive Explanations — a method to explain model predictions based on Shapley Values from game theory.   
We treat features as players in a cooperative game (players form coalitions which then can win some payout depending on the “strength” of the team), where the prediction is the payout. We try to divide the payout *fairly* between players. It is proven that Shapley values are the only *fair* distribution, where *fair* means that it satisfies some important axioms.

Thus, **with SHAP we explain how much each feature contributes to the value of a single prediction**. To be more precise, we explain how much it contributes to the deviation from the mean prediction of a chosen reference dataset. Further in the blog, you will see an example of such an explanation.

We can also aggregate SHAP values for many predictions to calculate different metrics, such as feature importance defined as the mean of absolute values of SHAP values.

The main advantage of SHAP values in contrast to other methods is a solid theory standing behind them.   
We can compute SHAP values for any model with zero knowledge of the model’s structure, and in such a general case, computational complexity would be exponential. So the computation time would be surely the biggest disadvantage, but there comes TreeSHAP…

**How does TreeSHAP work?**

TreeSHAP is an algorithm to compute SHAP values for tree ensemble models such as decision trees, random forests, and gradient boosted trees in a polynomial-time proposed by Lundberg et. al (2018)¹.

The algorithm allows us to reduce the complexity from O(TL2^M)to O(TLD^2) (T = number of trees in the model, L = maximum number of leaves in the tree, D = maximum depth of a tree, M = number of explained features). We can do this thanks to the structure of tree-based models and properties of Shapley values, mainly additivity (meaning that SHAP value for the model being a forest is a sum of SHAP values for all its trees).   
To further ensure that our method works fast, R package treeshap integrates C++ implementation of the algorithm.

TreeSHAP was originally implemented as a part of Python package shap .  
In the past, as MI2DataLab we have developed an R wrapper of this library — shapper, but it is a less stable and convenient solution than a standalone package.

treeshap works in speed comparable to the aforementioned Python library. Here we can spoil that in the future we will work on improving complexity to O(TLD) which may allow treeshap to outperform shap. The package also implements other features like different plotting functions.

Now, let’s have a look at the example showing how to use it.

**How to use it?**

Let’s now check how it works in practice. For our examples, we will use the apartments dataset that is available in the DALEX package. It contains artificially generated information about apartments in Warsaw. Currently, treeshap doesn’t work with factor features so we one-hot encode them with function from the mlr package.

library(treeshap)

aps\_data <- mlr::createDummyFeatures(DALEX::apartments)

As was mentioned above, the treeshap package works for various tree ensemble models, however, for the purposes of today’s examples, we will use random forest implementation from the ranger package to estimate the price for the square meter of an apartment.

model <- ranger::ranger(m2.price ~ ., data = aps\_data)

We have to start with unifying the model. That operation changes the object of the tree ensemble model into a form understandable for treeshap. That unified form makes it possible to smoothly compute Shapley values. The unified form is also easier to be interpreted by the user and can be used to predict with it using predict function. It is required to pass data along with the dataset on which we want to calculate Shapley values. Keep in mind that right now treeshap works only for numeric values of features. Categorical variables have to be encoded before training the model.

model\_unified <- ranger.unify(model, aps\_data)

Dataset used as reference (deviation from its mean prediction is explained) for which explanations are being computed can be changed at any time using set\_reference\_dataset function:

unified2 <- set\_reference\_dataset(model\_unified, aps\_data[1:2000, ])

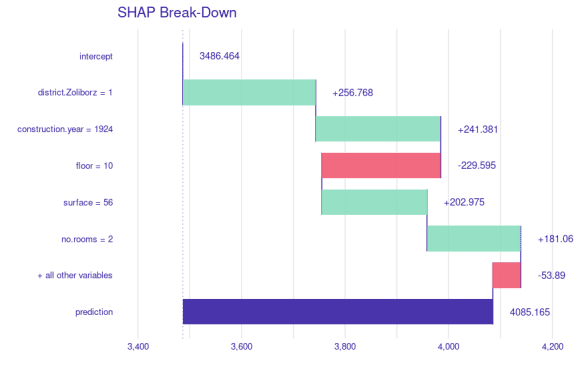
With a unified model on the board, we can now actually compute SHAP values. Raw SHAP values in the form of dataframe can be accessed as$shaps element of the result object.

treeshap\_res <- treeshap(model\_unified, aps\_data[1:500, ])

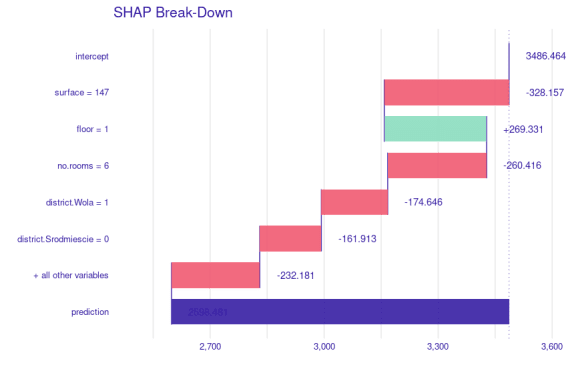
treeshap\_res$shaps

SHAP values can be used to explain contribution of features into the prediction for a single observation.

plot\_contribution(treeshap\_res, obs = 234, min\_max = c(3400, 4200))

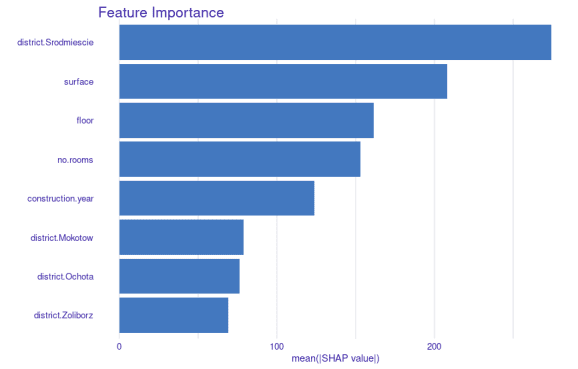


plot\_contribution(treeshap\_res, obs = 235, min\_max = c(2500, 3600))



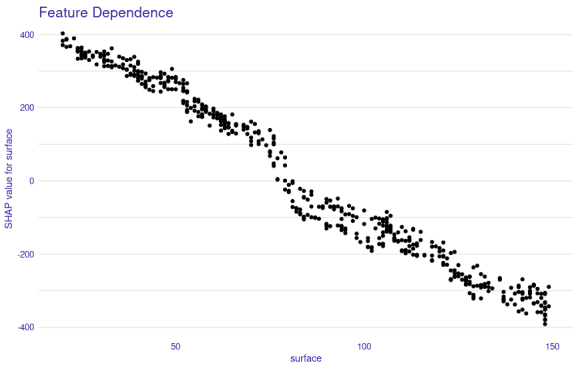
We can aggregate SHAP values from the dataset or its parts to acquire SHAP-based feature importance.

plot\_feature\_importance(treeshap\_res, max\_vars = 8)



Here we can see that whether the apartment is localized in Śródmieście, which is the center of Warsaw, has a huge impact on the price. There is a big gap between it and other districts. Śródmieście is highly preferred and hence the difference. The second most important feature is the surface. We can profile the variable to learn more about it.

plot\_feature\_dependence(treeshap\_res, 'surface')



As the surface of the flat goes up, the price for the squared meter consistently goes down, which is reasonable.

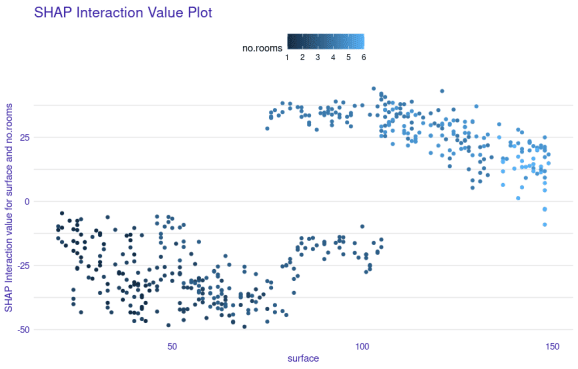
**Interactions**

SHAP interaction values are simply SHAP values for two-feature interactions. Calculation of them does not differ much from standard Shapley values. It requires only setting interactionsparameter in treeshap function and you can enjoy your brand new interactions without an estimated time of 2 years to compute.

inter <- treeshap(model\_unified, aps\_data[1:500,], interactions = T)

Computation of interactions can take a few minutes but it is still very fast and we can track progress on a progress bar. Raw SHAP interaction values in the form of an array can be accessed as $interactionselement of the result object. The array is a M by M matrix for every observation. Moreover, we can visualize those interactions! Isn’t it amazing?

plot\_interaction(treeshap\_interactions, 'surface', 'no.rooms')



**Try it yourself!**